Generating Training Data for Learning Linear Composite Dispatching Rules for Scheduling

Abstract

A supervised learning approach to generating composite linear priority dispatching rules for scheduling is studied. In particular we investigate a number of strategies for generating training data for learning a linear dispatching rule using preference learning. The results show that generating training data set from optimal solutions only is not as effective as when suboptimal solutions are added to the set. Furthermore, different strategies for creating preference pairs is investigated as well as suboptimal solution trajectories. The different strategies are investigated on 2000 randomly generated problem instances using two different problems generator settings.

When applying learning algorithms, the training set is of paramount importance. Training set should have sufficient knowledge of the problem at hand. This is done by the use of features, which are supposed to capture the essential measures of a problem's state. For this purpose, the job-shop scheduling problem (JSP) is used as a case study to illustrate a methodology for generating a meaningful training data, which can be successfully learned.

JSP deals with the allocation of tasks of competing resources where the goal is to minimise a schedule's maximum completion time, i.e. the makespan denoted $C_{\max}$. In order to find good solutions, heuristics are commonly applied in research, such as the simple priority based dispatching rules (SDR) from (Panwalkar and Iskander 1977). Composites of such simple rules can perform significantly better (Jayamohan and Rajendran 2004). As a consequence a linear composite of dispatching rules (LCDR) was presented in (2011). The goal there was to learn a set of weights, $\mathbf{w}$, via logistic regression such that

$$h(\mathbf{x}_j) = \left\langle \mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x}_j) \right\rangle, \tag{1}$$

yields the preference estimate for dispatching job j that corresponds to post-decision state $\mathbf{x}_j$, where $\boldsymbol{\phi}(\mathbf{x}_j)$ denotes its feature mapping. The job dispatched is the following,

$$j^* = \arg\max_j \left\{ h(\mathbf{x}_j) \right\}. \tag{2}$$

The approach was to use supervised learning, to determine which feature states are preferable to others. The training data was created from optimal solutions of randomly generated problem instances.

An alternative would be minimising the expected $C_{\max}$ directly using a brute force search such as CMA-ES (Hansen and Ostermeier 2001). Preliminary experiments were conducted in (2014), which showed that optimising the weights in eq. (1) via evolutionary search actually resulted in a better LCDR than the previous approach. The nature of the CMA-ES is to explore suboptimal routes until it converges to an optimal one. Implying that the previous approach of restricting the training data only to *one* optimal route may not produce a sufficiently rich training set. That is, the training set should incorporate a more complete knowledge on *all* possible preferences, i.e. make also the distinction between suboptimal and sub-suboptimal features, etc. This would require a Pareto ranking of preferences which can be used to make the distinction to which feature sets are equivalent, better or worse – and to what degree, e.g. by giving a weight to the preference. This would result in a very large training set, which of course could be re-sampled in order to make it computationally feasible to learn. In this study we will investigate a number of different ranking strategies for creating preference pairs.

Alternatively, training data could be generated using suboptimal solution trajectories. For instance (Li and Olafsson 2005) used decision trees to 'rediscover' largest processing time (LPT), a single priority based dispatching rule, by using LPT to create its training data. The limitations of using heuristics to label the training data is that the learning algorithm will mimic the original heuristic (both when it works poorly and well on the problem instances) and does not consider the real optimum. In order to learn heuristics that can outperform existing heuristics, then the training data needs to be correctly labelled. This drawback is confronted in (Malik et al. 2008; Russell et al. 2009; Olafsson and Li 2010) by using an optimal scheduler, computed off-line. In this study, we will both follow optimal and suboptimal solution trajectories, but for each partial solution the preference pair will be labelled correctly by solving the partial solution to optimality using a commercial software package (Gurobi Optimization, Inc. 2012). For this study most work remaining (MWR), a promising SDR for the given data distributions (Ingimundardottir and Runarsson 2012), and the CMA-ES optimised LCDRs from (In-

Table 1: Problem space distributions, $\mathcal{P}$, used in this study.

name	size ($n \times m$)	N_{train}	N_{test}	note
$\mathcal{P}_{jrnd}$	6×5	500	500	random
$\mathcal{P}_{jrndn}$	6×5	500	500	random-narrow

gimundardottir and Runarsson 2014) will be deemed worthwhile for generating suboptimal trajectories.

To summarise, the study considers two main aspects of the generation of training data: *a*) how preference pair are created at each decision stage, and *b*) which solution trajectorie(s) should be sampled. That is, optimal, random, or suboptimal ones, based on a good heuristic, etc.

This paper first illustrates how JSP can be seen as a decision tree where the depth of the tree corresponds total number of job-dispatches needed to form a complete schedule. The feature space is also introduced and how optimal dispatches and suboptimal dispatches are labelled at each node in the tree. This is followed by detailing the strategies investigated in this study by selecting preference pairs ranking and sampling solution trajectories. The authors then perform an extensive study comparing these strategies. Finally, this paper concludes with discussions and summary of main results.

Problem Space

In this study synthetic JSP data instances are considered with the problem size $n \times m$, where n and m denotes number of jobs and machines, respectively. Problem instances are generated stochastically by fixing the number of jobs and machines and processing time are i.i.d. and sampled from a discrete uniform distribution from the interval $I = [u_1, u_2]$, i.e. $p \sim \mathcal{U}(u_1, u_2)$. Two different processing times distributions are explored, namely $\mathcal{P}_{jrnd}$ where $I = [1, 99]$ and $\mathcal{P}_{jrndn}$ where $I = [45, 55]$, referred to as random and random-narrow, respectively. The machine order is a random permutation of all of the machines in the job-shop.

For each data distribution N_{train} and N_{test} problem instances were generated for training and testing, respectively. Values for N are given in table 1. Note, that difficult problem instances are not filtered out beforehand, such as the approach in (Watson et al. 2002).

JSP tree representation

When building a complete JSP schedule $\ell = n \cdot m$ dispatches must be made consecutively. A job is placed at the earliest available time slot for its next machine, whilst still fulfilling constraints that each machine can handle, which is at most one job at each time, and jobs need to have finished their previous machines according to its machine order. Unfinished jobs, referred to as the ready-list denoted $\mathcal{R}$, are dispatched one at a time according to some heuristic. After each dispatch the schedule's current features are updated based on its resulting partial schedule. For each possible post-decision state the temporal features applied in this study are given in table 2, which are based on SDRs which are widespread in practice. For example if $\mathbf{w}$ is zero, save for $w_6 = 1$,

Table 2: Feature space, $\mathcal{F}$, for $\mathcal{P}$ given the resulting temporal schedule after dispatching an operation.

$\boldsymbol{\phi}$	Feature description
ϕ_1	job processing time
ϕ_2	job start-time
ϕ_3	job end-time
ϕ_4	when machine is next free
ϕ_5	current makespan
ϕ_6	total work remaining for job
ϕ_7	most work remaining for all jobs
ϕ_8	total idle time for machine
ϕ_9	total idle time for all machines
ϕ_{10}	ϕ_9 weighted w.r.t. number of assigned tasks
ϕ_{11}	time job had to wait
ϕ_{12}	idle time created
ϕ_{13}	total processing time for job

then eq. (1) gives $h(\mathbf{x}_j) > h(\mathbf{x}_i)$, $\forall i$ which are jobs with less work remaining than job j, namely (2) yields the job with the highest ϕ_6 value, i.e. equivalent to dispatching rule most work remaining (MWR)

Figure 1 illustrates how the first two dispatches could be executed for a 6×5 JSP, with the machines, $a \in \{M_1, ..., M_5\}$, on the vertical axis and the horizontal axis yields the current makespan, $C_{\max}$. The next possible dispatches are denoted as dashed boxes with the job index j within and its length corresponding to processing time p_{ja}. In the top layer one can see an empty schedule. In the middle layer one of the possible dispatches from the layer above is fixed, depicted solid, and one can see the resulting schedule, i.e. what are the next possible dispatches given this new scenario? Finally, the bottom layer depicts all outcomes if job J_3 on machine M_4 would be dispatched. This sort of tree representation is similar to *game trees* (von Neumann and Morgenstern 2007) where the root node denotes the initial, i.e. empty, schedule and the leaf nodes denote the complete schedule, therefore the distance k from an internal node to the root yields the number of operations already dispatched. Traversing from root to leaf node one can obtain a sequence of dispatches that yielded the resulting schedule, i.e. the sequence indicates in which order the tasks should be dispatched for that particular schedule.

However, one can easily see that this sequence of task assignments is by no means unique. Inspecting a partial schedule further along in the dispatching process such as in fig. 1 (top layer), then let's say J_1 would be dispatched next, and in the next iteration J_2. Now this sequence would yield the same schedule as if J_2 would have been dispatched first and then J_1 in the next iteration, i.e. these are non-conflicting jobs. Which indicates that some of the nodes in the tree can merge. In the meantime the state of the schedules are different and thus their features, although they manage to merge with the same (partial) schedule at a later date. In this particular instance one can not infer that choosing J_1 is better and J_2 is worse (or vice versa) since they can both yield the same solution.

Furthermore, in some cases there can be multiple opti-

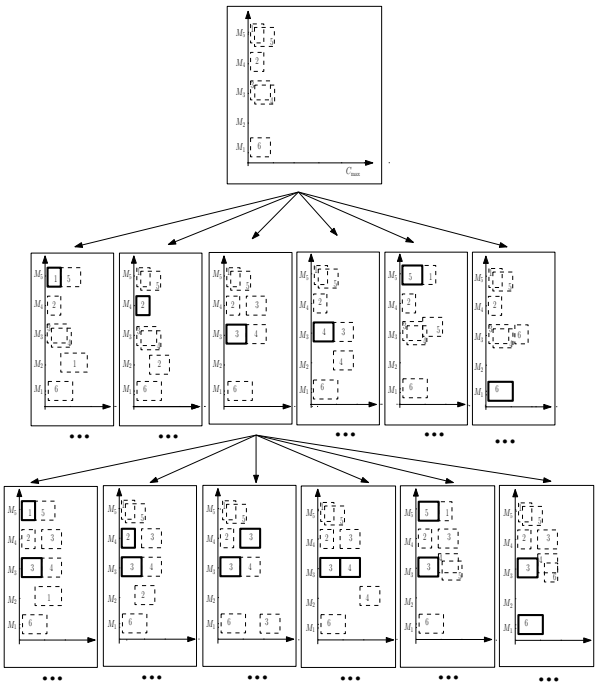


Figure 1: Partial Tree for JSP for the first two dispatches. Executed dispatches are depicted solid, and all possible dispatches are dashed.

mal solutions to the same problem instance. Hence not only is the sequence representation 'flawed' in the sense that slight permutations on the sequence are in fact equivalent w.r.t. the end-result, but varying permutations on the dispatching sequence (given the same partial initial sequence) can result in very different complete schedules but the same makespan, and thus same deviation from optimality, ρ defined by eq. (4), which is the measure under consideration. Care must be taken in this case that neither resulting features are labelled as undesirable or suboptimal. Only the resulting features from a dispatch resulting in a suboptimal solution should be labelled undesirable.

The creation of the tree for job-shop scheduling can be done recursively for all possible permutation of dispatches, in the manner described above, resulting in a full n-ary tree of height $\ell = n \cdot m$. Such an exhaustive search would yield at the most n^ℓ leaf nodes (worst case scenario being that no sub-trees merge). Now, since the internal vertices, i.e. partial schedules, are only of interest to learn,[1] the number of those can be at the most ${}^{n^\ell - 1}/_{n-1}$ (Rosen 2003). Even for small dimensions of n and m the number of internal vertices are quite substantial and thus computationally expensive to investigate them all.

The optimum makespan is known for each problem instance. At each time step (i.e. layer of the tree) a number of feature pair are created, they consist of the features $\boldsymbol{\phi}_o$ resulting from optimal dispatches $o \in \mathcal{O}^{(k)}$, versus features $\boldsymbol{\phi}_s$ resulting from suboptimal dispatches $s \in \mathcal{S}^{(k)}$ at time k. Note, $\mathcal{O}^{(k)} \cup \mathcal{S}^{(k)} = \mathcal{R}^{(k)}$ and $\mathcal{O}^{(k)} \cap \mathcal{S}^{(k)} = \emptyset$. In particular, each job is compared against another job of the ready-list, $\mathcal{R}^{(k)}$, and if the makespan differs, i.e. $C^{(s)}_{\max} \gneq C^{(o)}_{\max}$, an optimal/suboptimal pair is created, however if the makespan would be unaltered the pair is omitted since they give the same optimal makespan. This way, only features from a dispatch resulting in a suboptimal solution is labelled undesirable.

The approach taken in this study is to verify analytically, at each time step, whether it can indeed *somehow* yield an optimal schedule by manipulating the remainder of the sequence, yet maintaining the current temporal schedule fixed as its initial state. This also takes care of the scenario that having dispatched a job resulting in a different temporal makespan would have resulted in the same final makespan if another optimal dispatching sequence would have been chosen. That is to say the data generation takes into consideration when there are multiple optimal solutions to the same problem instance.

Selecting preference pairs

At each dispatch iteration k a number of preference pairs are created, which is then iterated over all N_{train} problem instance created. A separate data set is deliberately created for each dispatch iterations, as the initial feeling is that DRs used in the beginning of the schedule building process may not necessarily be the same as in the middle or end of the schedule. As a result there are ℓ linear scheduling rules for solving a $n \times m$ job-shop, specified by a set of preference pairs for each step,

$$S = \left\{ \{\boldsymbol{\phi}_o - \boldsymbol{\phi}_s, +1\}, \{\boldsymbol{\phi}_s - \boldsymbol{\phi}_o, -1\} \right\} \tag{3}$$

for all $o \in \mathcal{O}^{(k)}, s \in \mathcal{S}^{(k)}, k \in \{1, \ldots, \ell\}$. The reader is referred to (Ingimundardottir and Runarsson 2011) for a detailed description of how the linear ordinal regression model is trained on preference set S. Defining the size of the preference set as $l = |S|$, then if l is too large re-sampling may be needed to be done in order for the ordinal regression to be computationally feasible.

Trajectory sampling strategies

The following trajectory sampling strategies were explored for adding preference pairs to S,

S^{opt} at each dispatch some (random) optimal task is dispatched.

S^{cma} at each dispatch the task corresponding to highest priority, computed with fixed weights $\mathbf{w}$, which were obtained by directly optimising the mean of the performance measure, defined in eq. (4), with CMA-ES.

S^{mwr} at each dispatch the task corresponding to most work remaining is dispatched, i.e. following the simple dispatching rule MWR.

[1] The root is the empty initial schedule and for the last dispatch there is only one option left to dispatch, so there is no preferred 'choice' to learn.

S^{rnd} at each dispatch some random task is dispatched.

In the case of S^{mwr} and S^{cma} it is sufficient to explore each trajectory exactly once for each problem instance, since they are static DRs. Whereas, for S^{opt} and S^{rnd} there can be several trajectories worth exploring, however, only one is chosen (at random), this is deemed sufficient as the number of problem instances N_{train} is relatively large.

Ranking strategies

The following ranking strategies were implemented for adding preference pairs to S,

S_b all optimum rankings r_1 versus all possible sub-optimum rankings r_i, $i \in \{2, \ldots, n'\}$, preference pairs are added, i.e. same basic set-up as in (Ingimundardottir and Runarsson 2011).

S_f full subsequent rankings, i.e. all possible combinations of r_i and r_{i+1} for $i \in \{1, \ldots, n'\}$, preference pairs are added.

S_p partial subsequent rankings, i.e. sufficient set of combinations of r_i and r_{i+1} for $i \in \{1, \ldots, n'\}$, are added to the training set – e.g. in the cases that there are more than one operation with the same ranking, only one of that rank is needed to compared to the subsequent rank. Note that $S_p \subset S_f$.

where $r_1 > r_2 > \ldots > r_{n'}$ ($n' \leq n$) are the rankings of the ready-list, $\mathcal{R}^{(k)}$, at time step k.

Experimental study

To test the validity of different ranking and strategies, the problem spaces outlined in table 1 were used. The optimum makespan is denoted $C_{\max}^{\text{opt}}$, and the makespan obtained from the heuristic model by $C_{\max}^{\text{model}}$. Since the optimal makespan varies between problem instances the performance measure is the following,

$$\rho = \frac{C_{\max}^{\text{model}} - C_{\max}^{opt}}{C_{\max}^{\text{opt}}} \cdot 100\% \tag{4}$$

which indicates the percentage relative deviation from optimality.

The size of the preference set, S, for different trajectory and ranking strategies is depicted in fig. 2 for $\mathcal{P}_{jrnd}$ (above) and $\mathcal{P}_{jrndn}$ (below). The figure is divided vertically by problem space and horizontally by trajectory schemes.

A linear ordinal regression model (PREF) was created for each preference set, S, for problem spaces $\mathcal{P}_{jrnd}$ and $\mathcal{P}_{jrndn}$. A box-plot with the results of percentage relative deviation from optimality, ρ, defined by eq. (4), is presented in fig. 3. The boxplots are grouped w.r.t. trajectory schemes and color-coded w.r.t. ranking schemes. Moreover, the simple priority dispatching rule MWR and the weights obtained by the CMA-ES optimisation used to obtain the preference sets S^{mwr} and S^{cma} respectively, are shown in black in the far left of the group for comparison. From fig. 3 it is apparent there can be a performance edge gained by implementing a particular ranking or trajectory strategy, moreover the behaviour is analogous across different disciplines.

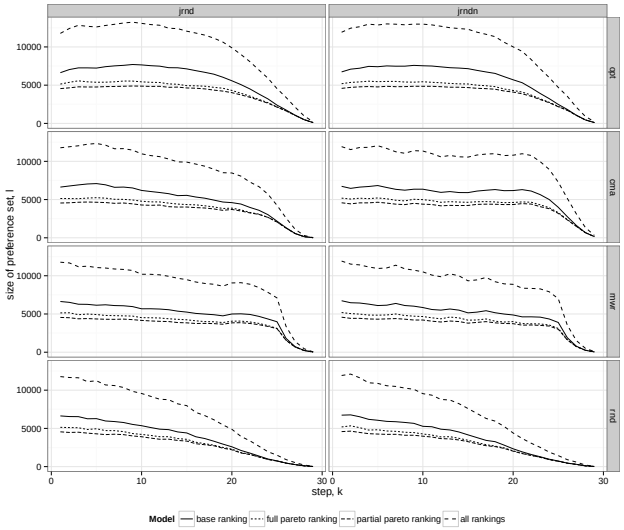


Figure 2: Size of preference set, l for different trajectory and ranking strategies, $\mathcal{P}_{jrnd}$ (left) and $\mathcal{P}_{jrndn}$ (right).

Main statistics are reported in tables 3a and 3b for $\mathcal{P}_{jrnd}$ and $\mathcal{P}_{jrndn}$, respectively. Models are sorted w.r.t. mean relative error.

Note that S^{all} denotes that all trajectories were explored, i.e. $S^{all} = S^{opt} \cup S^{cma} \cup S^{mwr} \cup S^{rnd}$. Similarly, S_a for all rankings, where $S_a = S_b \cup S_f \cup S_p$.

Ranking strategies

There is no statistical difference between PREF_f and PREF_p ranking-models across all trajectory disciplines (cf. Fig. 3), which is expected since S_p is designed to contain the same preference information as S_f. The results hold for both problem spaces.

Combining the ranking schemes, S_a, does not improve the individual ranking-schemes as there is no statistical difference between PREF_a and PREF_b, PREF_f nor PREF_p across all disciplines, save PREF_a^{cma} for $\mathcal{P}_{jrndn}$ which yielded a considerably worse mean relative error.

Moreover, there is no statistical difference between either of the Pareto ranking-schemes outperform and the original S_b set-up from (Ingimundardottir and Runarsson 2011). However overall, the Pareto-ranking schemes results in lower mean relative error, and since a smaller preference set is preferred, it is opted to use the S_p ranking scheme.

Furthermore, it is noted that PREF^{mwr} model is able to significantly outperform the original heuristics, MWR, used to create the training data S^{mwr}, irrespective of the ranking schemes. Whereas the fixed weights found via CMA-ES outperform the PREF^{cma} models for all ranking schemes. This implies that ranking scheme is relatively irrelevant. The results hold for both problem spaces.

Trajectory sampling strategies

Learning preference pairs from a good scheduling policies, as done in PREF^{cma} and PREF^{mwr}, can give favourable

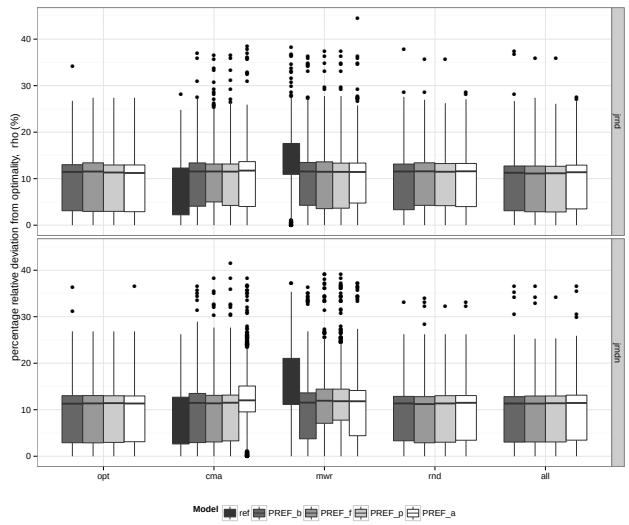


Figure 3: Box-plot of results for linear ordinal regression model trained on various preference sets using test sets for problem spaces $\mathcal{P}_{jrnd}$ (above) and $\mathcal{P}_{jrndn}$ (below).

results, however tracking optimal paths yields generally a lower mean relative error.

It is particularly interesting there is no statistical difference between PREF^{opt} and PREF^{rnd} for both $\mathcal{P}_{jrnd}$ and $\mathcal{P}_{jrndn}$ ranking-models. That is to say, tracking optimal dispatches gives the same performance as completely random dispatches. This indicates that exploring only optimal trajectories can result in a training set where the learning algorithm is inept to determine good dispatches in the circumstances when newly encountered features have diverged from the learned feature set labelled to optimum solutions.

Finally, PREF^{all} and PREF^{opt} gave the best combination for $\mathcal{P}_{jrnd}$ and $\mathcal{P}_{jrndn}$, however in the latter case PREF^{rnd} had the best mean relative error although not statistically different from PREF^{all} and PREF^{opt}.

For $\mathcal{P}_{jrnd}$ the best mean relative error was for PREF^{all}, in that case adding random suboptimal trajectories with the optimal trajectories gave the learning algorithm a greater variety of preference pairs for getting out of local minima. Therefore a general trajectory scheme would to explore both optimal with suboptimal paths.

Following CMA-ES guided trajectory

The rational for using the S^{cma} strategy was mostly due to the fact a linear classifier is creating the training data (using the weights found via CMA-ES optimisation), hence the training data created should be linearly separable, which in turn should boost the training accuracy for a linear classification learning model. However, this is not the case, since PREF^{cma} does not improve the original CMA-ES heuristic which was used to guide its preference set S^{cma}. However the PREF^{cma} approach is preferred to that of PREF^{mwr}, so there is some information gained by following the CMA-ES obtained weights instead of simple priority

Table 3: Main statistics of percentage relative deviation from optimality, ρ, defined by (4) for various models.

(a) $\mathcal{P}_{jrnd}$ test set

model	track	rank	mean	med	sd	min	max
CMA			8.84	10.59	6.14	0.00	28.18
PREF	all	p	9.63	11.16	6.32	0.00	35.97
PREF	all	f	9.68	11.11	6.38	0.00	35.97
PREF	opt	a	9.92	11.22	6.49	0.00	27.39
PREF	all	b	9.98	11.27	6.61	0.00	37.36
PREF	opt	b	10.05	11.45	6.53	0.00	34.23
PREF	opt	p	10.13	11.33	6.74	0.00	27.39
PREF	all	a	10.15	11.38	6.30	0.00	27.57
PREF	opt	f	10.31	11.54	6.87	0.00	27.39
PREF	rnd	b	10.51	11.55	6.86	0.00	37.87
PREF	rnd	p	10.75	11.49	6.70	0.00	35.60
PREF	cma	p	10.78	11.52	6.89	0.00	36.60
PREF	rnd	a	10.82	11.59	6.73	0.00	28.65
PREF	cma	f	10.90	11.55	6.89	0.00	36.60
PREF	cma	b	10.90	11.55	7.10	0.00	36.91
PREF	mwr	p	10.95	11.46	7.26	0.00	37.47
PREF	mwr	f	11.07	11.48	7.35	0.00	37.47
PREF	rnd	f	11.09	11.58	6.92	0.00	35.60
PREF	mwr	a	11.09	11.44	7.21	0.00	44.55
PREF	mwr	b	11.30	11.54	7.63	0.00	36.26
PREF	cma	a	11.39	11.74	7.59	0.00	38.38
MWR			13.76	12.72	7.41	0.00	38.27

(b) $\mathcal{P}_{jrndn}$ test set

model	track	rank	mean	med	sd	min	max
CMA			9.13	10.91	6.16	0.00	26.23
PREF	rnd	b	9.82	11.36	6.07	0.00	33.05
PREF	rnd	f	9.87	11.22	6.57	0.00	33.92
PREF	opt	b	9.94	11.31	6.52	0.00	36.32
PREF	opt	f	9.98	11.36	6.58	0.00	26.84
PREF	rnd	p	9.99	11.35	6.42	0.00	32.33
PREF	opt	a	10.01	11.34	6.31	0.00	36.60
PREF	all	f	10.05	11.33	6.53	0.00	36.60
PREF	opt	p	10.06	11.42	6.52	0.00	26.84
PREF	all	p	10.08	11.39	6.49	0.00	34.15
PREF	all	b	10.12	11.34	6.73	0.00	36.60
PREF	rnd	a	10.14	11.49	6.25	0.00	33.05
PREF	all	a	10.39	11.45	6.69	0.00	36.60
PREF	cma	f	10.56	11.38	7.28	0.00	38.31
PREF	cma	b	10.73	11.47	7.62	0.00	36.60
PREF	cma	p	10.74	11.51	7.43	0.00	41.60
PREF	mwr	b	11.33	11.52	7.72	0.00	36.41
PREF	mwr	a	11.70	11.82	7.88	0.00	37.20
PREF	mwr	f	12.07	11.93	8.07	0.00	39.17
PREF	mwr	p	12.14	11.84	8.32	0.00	39.12
PREF	cma	a	12.59	12.02	7.94	0.00	38.27
MWR			14.16	12.74	7.59	0.00	37.25

dispatching rules, such as MWR. Let's inspect the CMA-ES guided training data more closely, in particular the linear weights for eq. (1). The weights are depicted in fig. 4 for problem spaces $\mathcal{P}_{jrnd}$ (above) and $\mathcal{P}_{jrndn}$ (below). The original weights found via CMA-ES optimisation, that are used to guide the collection of training data, are depicted dashed whereas weights obtained by the linear classification PREF$^{cma}_{p}$ model are depicted solid.

From the CMA-ES experiments it is clear that a lot of weight is applied to decision variable w_6, which corresponds implementing MWR, yet the existing weights for other features directs the evolutionary search to a "better" training data to learn, than the PREF models. Arguably, the training data could be even better, however implementing CMA-ES is rather costly, in fact in (Ingimundardottir and Runarsson 2014) the optimisation had not fully converged given its allocated 288hrs of computation time.

It might also be an artefact due to the fact that the sampling of the feature space during CMA-ES search is completely different to the data generation described in this study. Hence the different scaling parameters for the features might influence the results. Moreover, the CMA-ES is minimizing the makespan directly, whereas the PREF models are learning to discriminate optimal versus suboptimal features sets that are believed to imply a better deviation from optimality later on. However, in that case, the process is very vulnerable when it comes to any divergence from the optimal path. Ideally, it would be best to combine both methodologies: Collect training data from the CMA-ES optimisation which optimises w.r.t. the ultimate performance measure used, and in order to improve upon those weights even further, use a preference based learning approach to deter from any local minima.

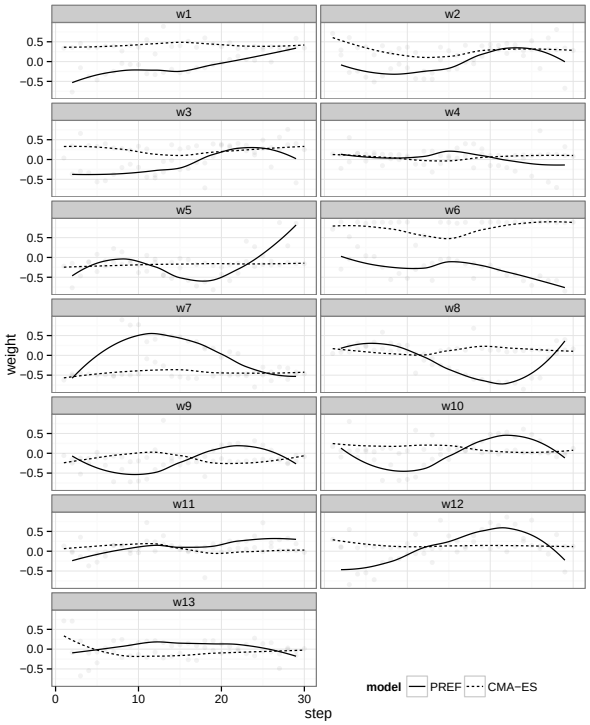
Summary and conclusion

The study presents strategies for how to generate training data to be used in supervised learning of linear composite dispatching rules for job shop scheduling. The experimental results provide evidence of the benefit of adding suboptimal solutions to the training set, apart from optimal ones. The classification of optimal[2] and suboptimal features are of paramount importance. The subsequent rankings are not of much value, since they are disregarded anyway. However, the trajectories to create training instances have to be varied.

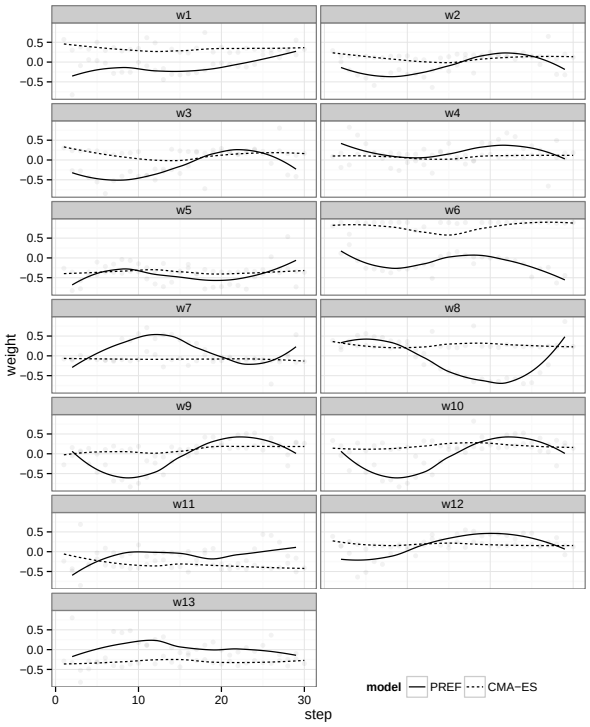
Unlike (Olafsson and Li 2010; Malik et al. 2008; Russell et al. 2009) learning only optimal training data was not fruitful. However, inspired by the original work by (Li and Olafsson 2005), having heuristic guide the generation of training data, but with nevertheless optimal labelling based on a solver, gave meaningful preference pairs which the learning algorithm could learn. In conclusion, henceforth, the training data will be generated with S_p^{all} scheme for the authors' future work.

Based on these preliminary experiments, we continue to test on a greater variety of problem data distributions for scheduling, namely job-shop and permutation flow-shop problems. Once training data has been carefully created, global dispatching rules can finally be learned, with the hope of implementing them for a greater number of jobs and machines. This is the focus of our current work.

[2]Here the tasks labelled 'optimal' do not necessarily yield the optimum makespan (except in the case of following optimal trajectories), instead these are the optimal dispatches for the given partial schedule.



(a) $\mathcal{P}_{jrnd}$



(b) $\mathcal{P}_{jrndn}$

Figure 4: Linear weights (w_1 to w_{13} from left to right, top to bottom) found via CMA-ES optimisation (dashed), and weights found via learning classification PREF$_p^{cma}$ model (solid).

References

Gurobi Optimization, Inc. 2012. Gurobi optimization (version 5.0) [software].

Hansen, N., and Ostermeier, A. 2001. Completely derandomized self-adaptation in evolution strategies. *Evol. Comput.* 9(2):159–195.

Ingimundardottir, H., and Runarsson, T. P. 2011. Supervised learning linear priority dispatch rules for job-shop scheduling. In Coello, C., ed., *Learning and Intelligent Optimization*, volume 6683 of *Lecture Notes in Computer Science*. Springer Berlin, Heidelberg. 263–277.

Ingimundardottir, H., and Runarsson, T. P. 2012. Determining the characteristic of difficult job shop scheduling instances for a heuristic solution method. In Hamadi, Y., and Schoenauer, M., eds., *Learning and Intelligent Optimization*, Lecture Notes in Computer Science. Springer Berlin Heidelberg. 408–412.

Ingimundardottir, H., and Runarsson, T. P. 2014. Evolutionary learning of weighted linear composite dispatching rules for scheduling.

Jayamohan, M., and Rajendran, C. 2004. Development and analysis of cost-based dispatching rules for job shop scheduling. *European Journal of Operational Research* 157(2):307–321.

Li, X., and Olafsson, S. 2005. Discovering dispatching rules using data mining. *Journal of Scheduling* 8:515–527.

Malik, A. M.; Russell, T.; Chase, M.; and Beek, P. 2008. Learning heuristics for basic block instruction scheduling. *Journal of Heuristics* 14(6):549–569.

Olafsson, S., and Li, X. 2010. Learning effective new single machine dispatching rules from optimal scheduling data. *International Journal of Production Economics* 128(1):118–126.

Panwalkar, S. S., and Iskander, W. 1977. A survey of scheduling rules. *Operations Research* 25(1):45–61.

Rosen, K. H. 2003. *Discrete Mathematics and Its Applications*. New York, NY, USA: McGraw-Hill, Inc., 5 edition. chapter 9, 631–700.

Russell, T.; Malik, A. M.; Chase, M.; and van Beek, P. 2009. Learning heuristics for the superblock instruction scheduling problem. *IEEE Trans. on Knowl. and Data Eng.* 21(10):1489–1502.

von Neumann, J., and Morgenstern, O. 2007. *Theory of Games and Economic Behavior (Commemorative Edition)*. Princeton Classic Editions. Princeton University Press.

Watson, J.-P.; Barbulescu, L.; Whitley, L. D.; and Howe, A. E. 2002. Contrasting structured and random permutation flow-shop scheduling problems: Search-space topology and algorithm performance. *INFORMS Journal on Computing* 14:98–123.